SQL Injection Attack Lab

Ethical Hacking 2022/23, University of Padua

Eleonora Losiouk, Alessandro Brighente, Gabriele Orazi, Francesco Marchiori

1 Overview

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers. Many web applications take inputs from users, and then use these inputs to construct SQL queries, so they can get information from the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When SQL queries are not carefully constructed, SQL injection vulnerabilities can occur. SQL injection is one of the most common attacks on web applications. In this lab, we have created a web application that is vulnerable to the SQL injection attack. Our web application includes the common mistakes made by many web developers. Students' goal is to find ways to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attack, and master the techniques that can help defend against such type of attacks. This lab covers the following topics:

- SQL statements: `SELECT` and `UPDATE` statements
- SQL injection
- Prepared statement

2 Lab Environment Setup

We have developed a web application for this lab, and we use containers to set up this web application. There are two containers in the lab setup, one for hosting the web application, and the other for hosting the database for the web application. The IP address for the web application container is `10.9.0.5`, and The URL for the web application is the following:

```
http://www.seed-server.com
```

We need to map this hostname to the container's IP address. Please add the following entry to the `/etc/hosts` file. You need to use the root privilege to change this file (using `sudo`). It should be noted that this name might have already been added to the file due to some other labs. If it is mapped to a different IP address, the old entry must be removed.

```
10.9.0.5    www.seed-server.com
```

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your machine (or in you VM, if you are using it) from Moodle, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. You can find more details and ways to resolve some problems in this manual.

In the following, we list some of the commonly used commands related to docker-compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (already configured in our provided VM, but you can easily add them to your local favourite `rc` file).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild   # Alias for: docker-compose build
$ dcup      # Alias for: docker-compose up
$ dcdown    # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC

$ dockps
b1004832e275    hostA-10.9.0.5
0af4ea7a3e2e    hostB-10.9.0.6
9652715c8e0a    hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#
```

Note that if a Docker command requires a container ID, you do not need to type the entire ID string. Typing the first few characters will be sufficient, as long as they are unique among all the containers.

If you want to use your local `rc` file, you can simply paste at the end the following:

```
# Aliases for the Docker Compose
alias dcbuild='sudo docker-compose build' # Alias for: docker-compose build
alias dcup='sudo docker-compose up' # Alias for: docker-compose up
alias dcdown='sudo docker-compose down' # Alias for: docker-compose down
alias dockps='sudo docker ps --format "{{.ID}} {{.Names}}"'
alias docksh='f(){ sudo docker exec -it $1 /bin/bash;  unset -f f; }; f'
```

Then, remember to relaunch the terminal or source your `rc` file.

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the `mysql_data` folder on the host machine (inside Labsetup, it will be created after the MySQL container runs once) to the `/var/lib/mysql` folder inside the MySQL container. This folder is where MySQL stores its database. Therefore, even if the container is destroyed, data in the database are still kept. If you do want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

2.2 About the Web Application

We have created a web application, which is a simple employee management application. Employees can view and update their personal information in the database through this web application. There are mainly two roles in this web application: `Administrator` is a privilege role and can manage each individual employees' profile information; `Employee` is a normal role and can view or update his/her own profile information. All employee information is given in the following.

```
---------------------------------------------------------------------------
Name    | Employee ID   | Password  | Salary    | Birthday  | SSN        |
---------------------------------------------------------------------------
Admin   | 99999         | seedadmin | 400000    | 3/5       | 43254314   |
Alice   | 10000         | seedalice | 20000     | 9/20      | 10211002   |
Boby    | 20000         | seedboby  | 50000     | 4/20      | 10213352   |
Ryan    | 30000         | seedryan  | 90000     | 4/10      | 32193525   |
Samy    | 40000         | seedsamy  | 40000     | 1/11      | 32111111   |
Ted     | 50000         | seedred   | 110000    | 11/3      | 24343244   |
---------------------------------------------------------------------------

-----------------------------------------------
  | Nickname  | Email | Address   | Phone#    |
-----------------------------------------------
  |           |       |           |           |
  |           |       |           |           |
  |           |       |           |           |
  |           |       |           |           |
  |           |       |           |           |
  |           |       |           |           |
-----------------------------------------------
```

3 Lab Tasks

3.1 Task 1: Get Familiar with SQL Statements

The objective of this task is to get familiar with SQL commands by playing with the provided database. The data used by our web application is stored in a MySQL database, which is hosted on our MySQL container. We have created a database called `sqllab_users`, which contains a table called `credential`. The table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee. In this task, you need to play with the database to get familiar with SQL queries.

Please get a shell on the MySQL container (see the container manual for instruction; the manual is linked to the lab's website). Then use the `mysql` client program to interact with the database. The user name is `root` and password is `dees`.

```
// Inside the MySQL container
# mysql -u root -pdees
```

After login, you can create new database or load an existing one. As we have already created the `sqllab_users` database for you, you just need to load this existing database using the `use` command. To show what tables are there in the `sqllab_users` database, you can use the `show tables` command to print out all the tables of the selected database.

```
mysql> use sqllab_users;
Database changed
```

```
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
```

After running the commands above, you need to use a SQL command to print all the profile information of the employee Alice. Please provide the screenshot of your results.

3.2 Task 2: SQL Injection Attack on SELECT Statement

SQL injection is basically a technique through which attackers can execute their own malicious SQL statements generally referred as malicious payload. Through the malicious SQL statements, attackers can steal information from the victim database; even worse, they may be able to make changes to the database. Our employee management web application has SQL injection vulnerabilities, which mimic the mistakes frequently made by developers.

We will use the login page from www.seed-server.com for this task. The login page is shown in Figure 1. It asks users to provide a user name and a password. The web application authenticate users based on these two pieces of data, so only employees who know their passwords are allowed to log in. Your job, as an attacker, is to log into the web application without knowing any employee's credential.

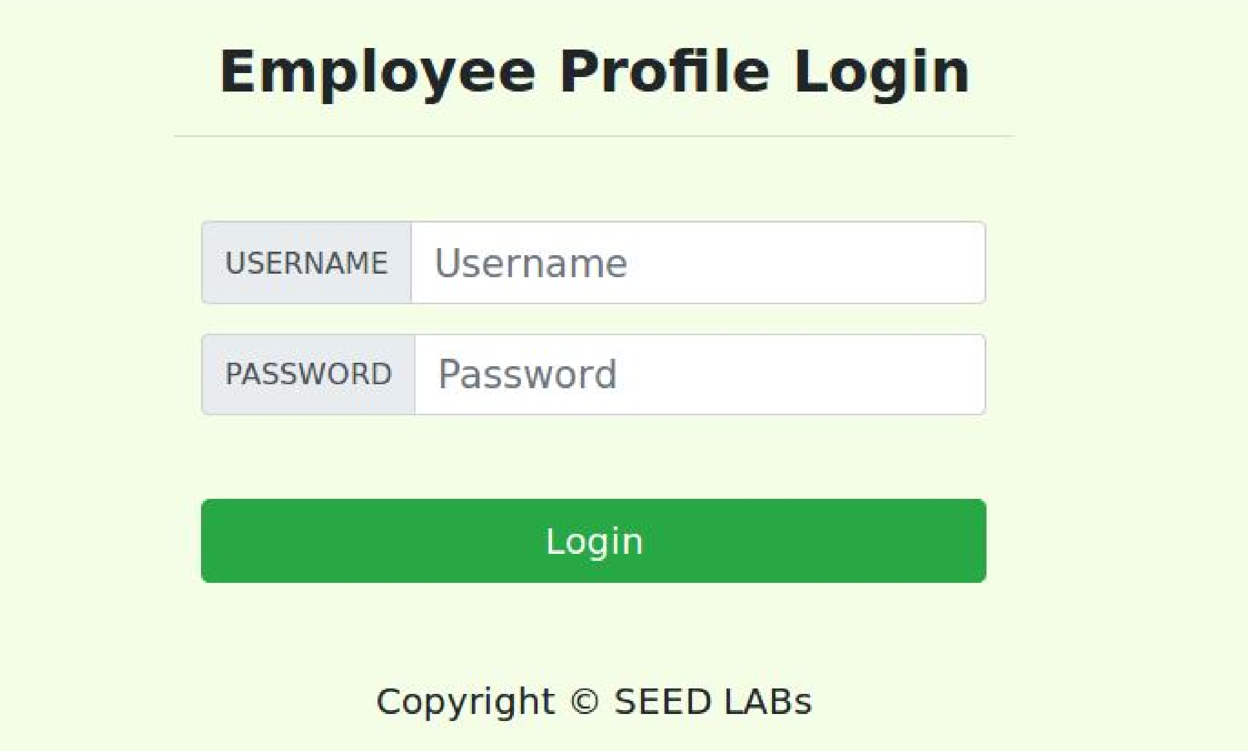

Figure 1: The Login page

To help you started with this task, we explain how authentication is implemented in the web application. The PHP code `unsafe_home.php`, located in the `/var/www/SQL_Injection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

```
$input_uname = $_GET[’username’];
$input_pwd = $_GET[’Password’];
$hashed_pwd = sha1($input_pwd);
```

```
...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
            nickname, Password
        FROM credential
        WHERE name= ’$input_uname’ and Password=’$hashed_pwd’";
$result = $conn -> query($sql);

// The following is Pseudo Code
if(id != NULL) {
    if(name==’admin’) {
        return All employees information;
    } else if (name !=NULL){
        return employee information;
    }
} else {
    Authentication Fails;
}
```

The above SQL statement selects personal employee information such as id, name, salary, ssn etc from the `credential` table. The SQL statement uses two variables `input_uname` and `hashed_pwd`, where `input_uname` holds the string typed by users in the username field of the login page, while `hashed_pwd` holds the `sha1` hash of the password typed by the user. The program checks whether any record matches with the provided username and password; if there is a match, the user is successfully authenticated, and is given the corresponding employee information. If there is no match, the authentication fails.

Task 2.1: SQL Injection Attack from webpage. Your task is to log into the web application as the administrator from the login page, so you can see the information of all the employees. We assume that you do know the administrator's account name which is admin, but you do not the password. You need to decide what to type in the `Username` and `Password` fields to succeed in the attack.

Task 2.2: SQL Injection Attack from command line. Your task is to repeat Task 2.1, but you need to do it without using the webpage. You can use command line tools, such as `curl`, which can send HTTP requests. One thing that is worth mentioning is that if you want to include multiple parameters in HTTP requests, you need to put the URL and the parameters between a pair of single quotes; otherwise, the special characters used to separate parameters (such as `&`) will be interpreted by the shell program, changing the meaning of the command. The following example shows how to send an HTTP GET request to our web application, with two parameters (`username` and `Password`) attached:

```
$ curl ’www.seed-server.com/unsafe_home.php?username=alice&Password=11’
```

If you need to include special characters in the `username` or `Password` fields, you need to encode them properly, or they can change the meaning of your requests. If you want to include single quote in those fields, you should use `%27` instead; if you want to include white space, you should use `%20`. In this task, you do need to handle HTTP encoding while sending requests using `curl`.

Task 2.3: Append a new SQL statement. In the above two attacks, we can only steal information from the database; it will be better if we can modify the database using the same vulnerability in the login page. An idea is to use the SQL injection attack to turn one SQL statement into two, with the second one being the update or delete statement. In SQL, semicolon (;) is used to separate two SQL statements. Please try to run two SQL statements via the login page. There is a countermeasure preventing you from running two SQL statements in this attack. Please figure out what this countermeasure is, and describe your discovery in the lab report.

3.3 Task 3: SQL Injection Attack on UPDATE Statement

If a SQL injection vulnerability happens to an UPDATE statement, the damage will be more severe, because attackers can use the vulnerability to modify databases. In our Employee Management application, there is an Edit Profile page (Figure 2) that allows employees to update their profile information, including nickname, email, address, phone number, and password. To go to this page, employees need to log in first.

When employees update their information through the Edit Profile page, the following SQL UPDATE query will be executed. The PHP code implemented in `unsafe_edit_backend.php` file is used to update employee’s profile information. The PHP file is located in the `/var/www/SQLInjection` directory.

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
nickname=’$input_nickname’,
email=’$input_email’,
address=’$input_address’,
Password=’$hashed_pwd’,
PhoneNumber=’$input_phonenumber’
WHERE ID=$id;";
$conn->query($sql);
```

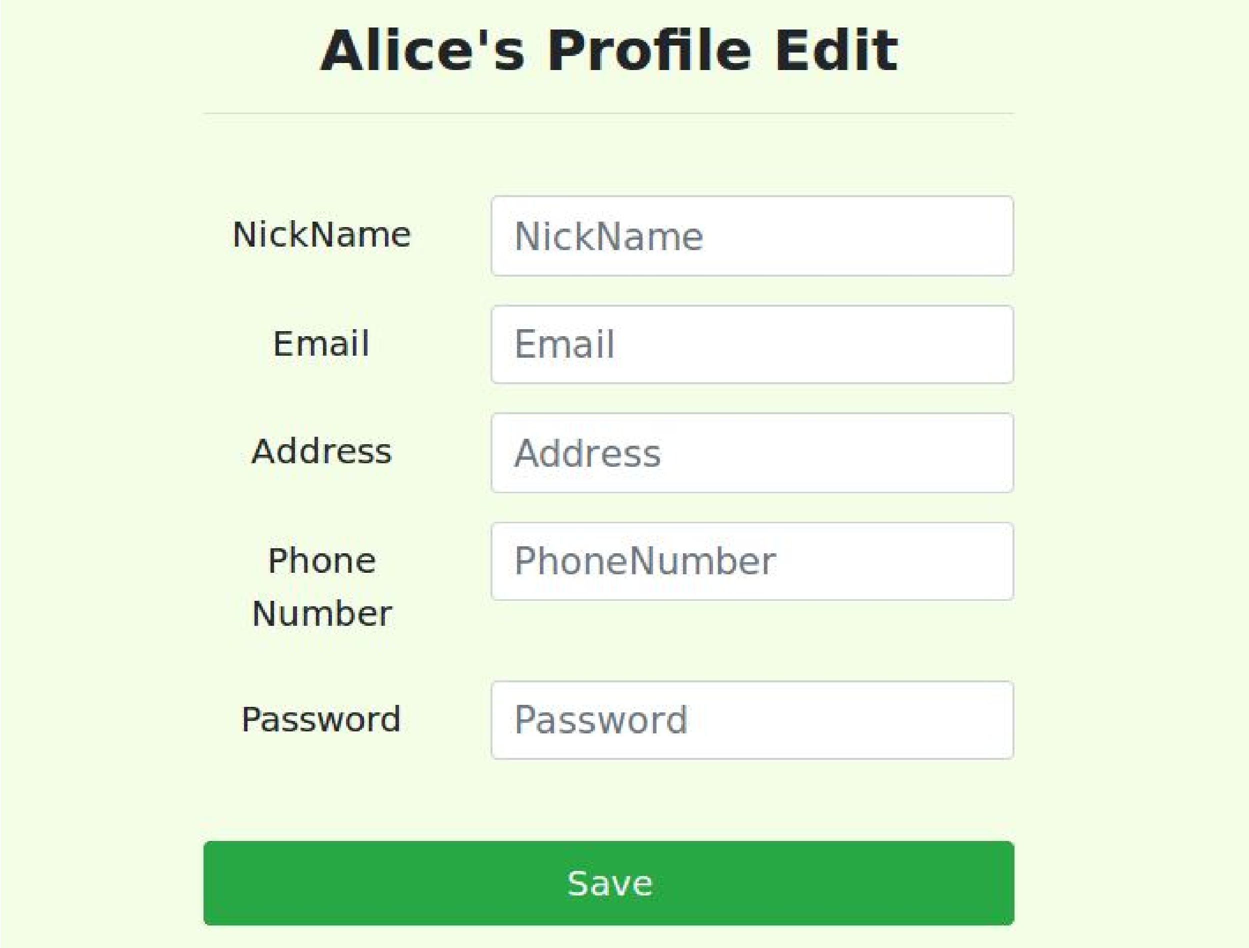

Figure 2: The Edit-Profile page

Task 3.1: Modify your own salary. As shown in the Edit Profile page, employees can only update their nicknames, emails, addresses, phone numbers, and passwords; they are not authorized to change their salaries. Assume that you (Alice) are a disgruntled employee, and your boss Boby did not increase your salary this year. You want to increase your own salary by exploiting the SQL injection vulnerability in the Edit-Profile page. Please demonstrate how you can achieve that. We assume that you do know that salaries are stored in a column called `salary`.

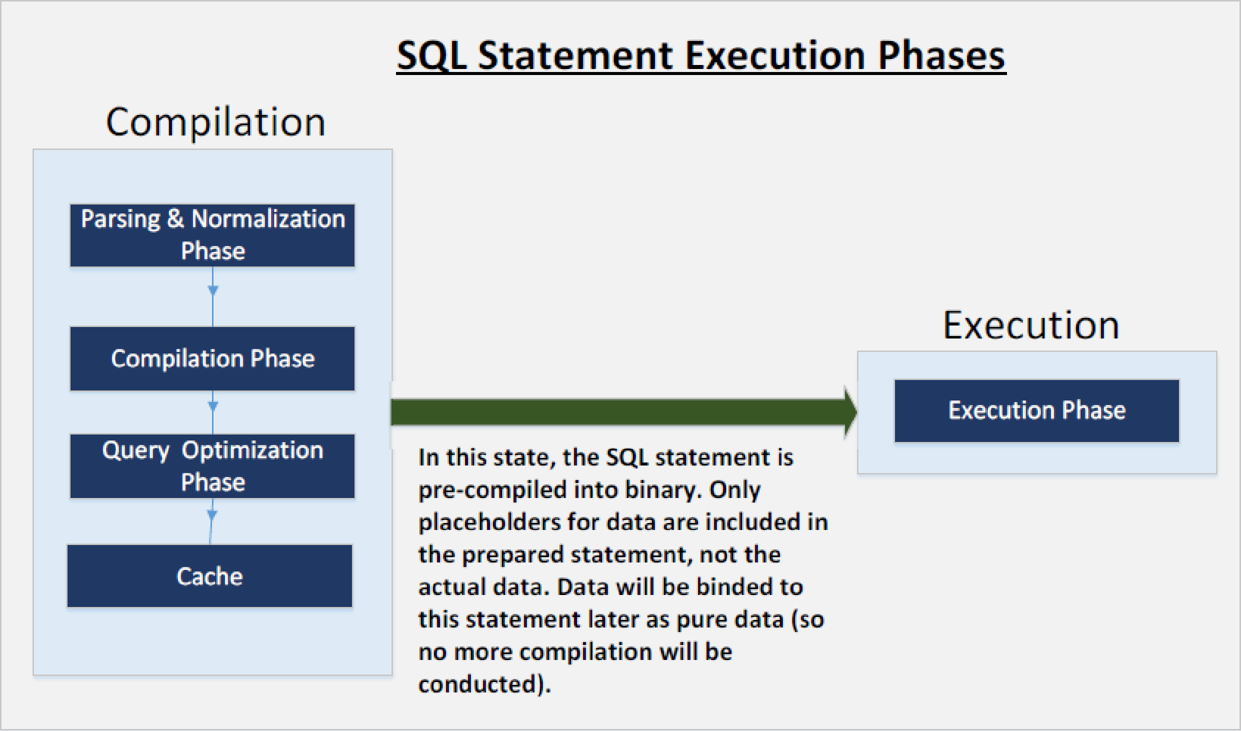
SQL Statement Execution Phases
Compilation
Parsing & Normalization Phase
Compilation Phase
Query Optimization Phase
Cache
Execution
Execution Phase
In this state, the SQL statement is pre-compiled into binary. Only placeholders for data are included in the prepared statement, not the actual data. Data will be binded to this statement later as pure data (so no more compilation will be conducted).

Figure 3: Prepared Statement Workflow

Task 3.2: Modify other people' salary. After increasing your own salary, you decide to punish your boss Boby. You want to reduce his salary to 1 dollar. Please demonstrate how you can achieve that.

Task 3.3: Modify other people' password. After changing Boby's salary, you are still disgruntled, so you want to change Boby's password to something that you know, and then you can log into his account and do further damage. Please demonstrate how you can achieve that. You need to demonstrate that you can successfully log into Boby's account using the new password. One thing worth mentioning here is that the database stores the hash value of passwords instead of the plaintext password string. You can again look at the `unsafe_edit_backend.php` code to see how password is being stored. It uses SHA1 hash function to generate the hash value of password.

3.4 Task 4: Countermeasure — Prepared Statement

The fundamental problem of the SQL injection vulnerability is the failure to separate code from data. When constructing a SQL statement, the program (e.g. PHP program) knows which part is data and which part is code. Unfortunately, when the SQL statement is sent to the database, the boundary has disappeared; the boundaries that the SQL interpreter sees may be different from the original boundaries that was set by the developers. To solve this problem, it is important to ensure that the view of the boundaries are consistent in the server-side code and in the database. The most secure way is to use prepared statement.

To understand how prepared statement prevents SQL injection, we need to understand what happens when SQL server receives a query. The high-level workflow of how queries are executed is shown in Figure 3. In the compilation step, queries first go through the parsing and normalization phase, where a query is checked against the syntax and semantics. The next phase is the compilation phase where keywords (e.g. SELECT, FROM, UPDATE, etc.) are converted into a format understandable to machines. Basically, in this phase, query is interpreted. In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is store in the cache, so whenever the next query comes in, it will be checked against the content in the cache; if it's already present in the cache, the parsing, compilation and query optimization phases will be skipped. The compiled query is then passed to the execution phase where it is actually executed.

Prepared statement comes into the picture after the compilation but before the execution step. A prepared statement will go through the compilation step, and be turned into a pre-compiled query with empty placeholders for data. To run this pre-compiled query, data need to be provided, but these data will not go through the compilation step; instead, they are plugged directly into the pre-compiled query, and are sent to the execution engine. Therefore, even if there is SQL code inside the data, without going through the compilation step, the code will be simply treated as part of data, without any special meaning. This is how prepared statement prevents SQL injection attacks.

Here is an example of how to write a prepared statement in PHP. We use a SELECT statement in the following example. We show how to use prepared statement to rewrite the code that is vulnerable to SQL injection attacks.

```
$sql = "SELECT name, local, gender
        FROM USER_TABLE
        WHERE id = $id AND password =’$pwd’ ";
$result = $conn->query($sql)
```

The above code is vulnerable to SQL injection attacks. It can be rewritten to the following:

```
$stmt = $conn->prepare("SELECT name, local, gender
                        FROM USER_TABLE
                        WHERE id = ? and password = ? ");
// Bind parameters to the query
$stmt->bind_param("is", $id, $pwd);
$stmt->execute();
$stmt->bind_result($bind_name, $bind_local, $bind_gender);
$stmt->fetch();
```

Using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps. The first step is to only send the code part, i.e., a SQL statement without the actual the data. This is the prepare step. As we can see from the above code snippet, the actual data are replaced by question marks (?). After this step, we then send the data to the database using `bind_param()`. The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement. In the `bind_param()` method, the first argument `"is"` indicates the types of the parameters: `"i"` means that the data in `$id` has the integer type, and `"s"` means that the data in `$pwd` has the string type.

Task. In this task, we will use the prepared statement mechanism to fix the SQL injection vulnerabilities. For the sake of simplicity, we created a simplified program inside the `defense` folder. We will make changes to the files in this folder. If you point your browser to the following URL, you will see a page similar to the login page of the web application. This page allows you to query an employee's information, but you need to provide the correct user name and password.

```
URL: http://www.seed-server.com/defense/
```

The data typed in this page will be sent to the server program `getinfo.php`, which invokes a program called `unsafe.php`. The SQL query inside this PHP program is vulnerable to SQL injection attacks. Your job is modify the SQL query in `unsafe.php` using the prepared statement, so the program can defeat SQL injection attacks. Inside the lab setup folder, the `unsafe.php` program is in the `image_www/Code/defense` folder. You can directly modify the program there. After you are done, you need to rebuild and restart the container, or the changes will not take effect.

You can also modify the file while the container is running. On the running container, the `unsafe.php` program is inside `/var/www/SQL_Injection/defense`. The downside of this approach is that in order to keep the docker image small, we have only installed a very simple text editor called `nano` inside the container. It should be sufficient for simple editing. If you do not like this editor, you can always use `"apt install"` to install your favoriate command-line editor inside the container. For example, for people who like vim, you can do the following:

```
# apt install -y vim
```

This installation will be discarded after the container is shutdown and destroyed. If you want to make it permanent, add the installation command to the `Dockerfile` inside the `image_www` folder.

4 Guidelines

Test SQL Injection String. In real-world applications, it may be hard to check whether your SQL injection attack contains any syntax error, because usually servers do not return this kind of error messages. To conduct your investigation, you can copy the SQL statement from php source code to the MySQL console. Assume you have the following SQL statement, and the injection string is `’ or 1=1;#`.

```
SELECT * from credential
WHERE name=’$name’ and password=’$pwd’;
```

You can replace the value of `$name` with the injection string and test it using the MySQL console. This approach can help you construct a syntax-error free injection string before launching the real attack.